

Session 5: Reading data code

Nested data, one-line loops, and
the two libraries all data science
in Python is built on.

Data Science and Advanced Programming · UNIL ·
2026

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9      a loop, in one line
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13            if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})
      a function, in one line
```

Code you will read every day

Ask a chatbot to analyse a portfolio, and the answer will look like the script on the left: data **inside** data, loops squeezed **into one line**, a function with **no name**.

Nothing on the left is new in principle: dicts, lists, loops and functions from sessions 3 and 4. What is new is **how compactly** they are written.

Today's goal is not to write this style fluently. It is to **read it**, calmly, piece by piece.

```
$ python prices.py
```

```
not run yet
```

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

Today: three scripts, one ledger

1. Nested data: data inside data
2. Comprehensions: loops in one line
3. `lambda` and other one-line tricks
4. `numpy`, properly: grids of numbers
5. `pandas`, properly: tables with names

Parts 1–3 use the script on the left. `numpy` and `pandas` each get a script of their own. All three value the same ledger.

```
$ python prices.py
```

```
not run yet
```

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13            if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})
```

Read it cold

Two minutes, on your own.

- Line 7 has **three** pairs of square brackets. What does each one pick?
- Lines 10 and 11 contain the word `for`, but no indented block. What might they do?
- **Guess** what line 16 prints.

```
$ python prices.py
```

```
not run yet
```

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

What it prints

You do not need to run anything now. Bottom left is the output of one run.

Seven lines. Session 3's numbers are all here: NESN at 102.3, the values 1905.0, 409.2 and 1024.0, and their total, 3338.2.

The same ledger, written the way experienced programmers, and AI assistants, write it.

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

REMINDER

Collections so far (sessions 3 and 4)

Type	Written	Look up by	Can change?
list	[185.0, 188.2]	position: p[0]	yes
tuple	("NESN", 4)	position	no
dict	{"shares": 4}	key: d["shares"]	yes
numpy array	np.array([10, 4, 2])	position	yes
pandas DataFrame	a table	column name	yes

Each is a **class** (session 4), with its own methods. Today we put them **inside each other**, and meet the last two properly.

PART 1

Nested data

Real data is rarely flat: lists inside dicts inside dicts.

prices.py

```
1 # prices.py - nested data, read in one line
2 ledger = { a dict of dicts, holding lists
3     "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4     "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5     "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6 }
7 print(ledger["NESN"]["prices"][-1])
8 print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

Data inside data

Lines 2–6 build **one** value: a dict whose values are dicts, one of which holds a list.

ledger		dict
— "AAPL"	→ {...}	dict
— "NESN"	→ {...}	dict
	— "shares" → 4	
	— "prices" → [104.1, 103.0, 102.3]	list
— "V00"	→ {...}	dict

Read it as a **tree**. Each level is a collection you already know, with its own way of looking things up.

prices.py

```
1 # prices.py – nested data, read in one line
2 ledger = {
3     "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4     "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5     "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6 }
7 print(ledger["NESN"]["prices"][-1])
8 print(ledger["NESN"]["prices"][-1])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

102.3 the last NESN price

```
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

Reading a chain of look-ups

Read from **left to right**. Each `[ ]` is applied to what the previous one returned:

line 7	<code>ledger["NESN"]["prices"][-1]</code>
	↓ <code>ledger["NESN"]</code> : a key of the outer dict
a dict	<code>{"shares": 4, "prices": [...]}</code>
	↓ <code>["prices"]</code> : a key of the inner dict
a list	<code>[104.1, 103.0, 102.3]</code>
	↓ <code>[-1]</code> : a position in the list
a float	<code>102.3</code>

At every step, ask: **what type do I have now?** The type decides what the next bracket may contain.

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])  # a slice at the end
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]  # days 1 and 2
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

Slicing at the end of a chain

A chain can end in a **slice** instead of an index. Line 8 takes AAPL's prices from day 1 onwards: `[188.2, 190.5]`.

Everything from session 3 works at the end of a chain: `[-1]`, `[1:]`, `[::-1]`, `len(...)`, `sum(...)`.

`sum(ledger["V00"]["prices"])` adds VOO's three prices; `len(ledger["AAPL"]["prices"])` is 3.

prices.py

```
1 # prices.py - nested data, read in one line
2 ledger = {
3     "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4     "NESN": {"shares": 4, "prices": [102.3]},
5     "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6 }
7 print(ledger["NESN"]["prices"][-1])
8 print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

REMINDER

Indexing and slicing (session 3)

With `p = ledger["AAPL"]["prices"]`, that is `[185.0, 188.2, 190.5]`:

You write	Gives	Read it as
<code>p[0], p[-1]</code>	<code>185.0, 190.5</code>	first, last
<code>p[1:]</code>	<code>[188.2, 190.5]</code>	from position 1 to the end
<code>p[:2]</code>	<code>[185.0, 188.2]</code>	up to, not including, 2
<code>p[-2:]</code>	<code>[188.2, 190.5]</code>	the last two
<code>p[::2]</code>	<code>[185.0, 190.5]</code>	every second one
<code>p[::-1]</code>	<code>[190.5, 188.2, 185.0]</code>	backwards

`[start:stop:step]` : any part may be left out. An index gives **one item**; a slice gives **a list**.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["price"])  # no key "price"
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13            if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})

```

A wrong key, deep inside

Line 7 now asks for `"price"`: the key is `"prices"`. The traceback's carets point at **which** bracket failed: the second one.

`KeyError: 'price'` names the key that is missing. With nested data, check **each level**: `print(ledger.keys())`, then `print(ledger["NESN"].keys())`.

```
$ python prices.py
```

```

Traceback (most recent call last):
  File "prices.py", line 7, in <module>
    print(ledger["NESN"]["price"])
    ~~~~~~^~~~~~
KeyError: 'price'

```

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][3])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13            if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})

```

only 0, 1, 2

⚠ WARNING

One error per level

Now line 7 asks for position 3 of a three-price list:
 IndexError.

Level	Wrong	Error
dict	a missing key	KeyError
list	a position too far	IndexError
a value of the wrong type	ledger["NESN"][0]	KeyError: 0 (a dict has no positions)

The error type tells you **which kind of level** you were at when it broke.

```
$ python prices.py
```

```

Traceback (most recent call last):
  File "prices.py", line 7, in <module>
    print(ledger["NESN"]["prices"][3])
    ~~~~~^~^
IndexError: list index out of range

```

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13            if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

Looping over nested data

`.items()` (session 3) gives each **key** with its **value**.
Here each value is itself a dict:

Pass	t	d
1	"AAPL"	{"shares": 10, "prices": [...]}
2	"NESN"	{"shares": 4, "prices": [...]}
3	"V00"	{"shares": 2, "prices": [...]}

Inside the loop, `d["prices"][-1]` is one position's last price. Line 10 does this for every position, in one line:
Part 2.

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

+ GOOD TO KNOW

This is what JSON looks like

Data from the web, from APIs, and from many files arrives as **JSON**: nested dicts and lists, written almost exactly like lines 2–6.

```
{"AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]}, ...}
```

Python's `json` module turns such text into these dicts and lists in one call. The skill you practised in this part, reading a chain of look-ups level by level, is the whole skill of working with JSON.

PART 2

Comprehensions

A loop that builds a list, written in one line.

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [d["prices"][-1] > d["prices"][0] for t, d in ledger.items()]
13 print(last)
14 print(values, sum(values))
15 print(rising)
16
17 by_value = sorted(ledger, reverse=True,
18                   key=lambda t: ledger[t]["shares"] * last[t])
19 print(by_value)
20 change = lambda p: (p[-1] - p[0]) / p[0]
21 print({t: round(change(d["prices"]), 4)
22       for t, d in ledger.items()})
```

\$ python prices.py

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2 values
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

The accumulator pattern, squeezed

Session 3's "collect" pattern, and line 11, do the same thing:

```
a loop  values = []
        for t, d in ledger.items():
            values.append(d["shares"] * last[t])
line 11 values = [d["shares"] * last[t] for t, d in ledger.items()]
```

Same loop (blue), same value for each item (gold). The comprehension drops `values = []` and `.append`: the square brackets **are** the new list.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = []
12 for t, d in ledger.items():  < the loop version
13     values.append(d["shares"] * last[t])
14 print(last)
15 print(values, sum(values))
16 print(values == [1905.0, 409.2, 1024.0])
17
18 by_value = sorted(ledger, reverse=True,
19                    key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})

```

Prove it: the loop version on the left

Lines 11–13 now build `values` the long way, with a loop and `.append`. Output line 4 is unchanged, and line 16 confirms it: `True`.

Any comprehension can be rewritten as a loop, and back. When a comprehension confuses you, **rewrite it as a loop** in your head, or on paper.

```

$ python prices.py
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2  < same list
True
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}

```

prices.py

```
1  # prices.py - nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: 2. then this ')[-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items() if d["prices"][-1] > d["prices"][0]]
13         1. read this first
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

How to read a comprehension

```
values = [ d["shares"] * last[t] for t, d in ledger.items() ]
```

1. the loop: what goes round? 2. the expression: what is kept each time?

1. Find **for**. Read **from** **for** **to the end**: "for each ticker **t** and inner dict **d** in the ledger..."
2. Go back to the **start**: "...keep **d["shares"] * last[t]**."
3. Look at the **brackets**: **[]** makes a list.

Say it in one sentence: **"the value of each position, as a list."**

prices.py

```
1  # prices.py - nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13            if d["prices"][-1] > d["prices"][0]] 3. a filter
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                    key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00'] NESN fell: left out
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

A comprehension with a filter

```
rising = [t for t, d in ledger.items()
           if d["prices"][-1] > d["prices"][0]]
```

1. loop 2. keep only if... 3. what to keep

An `if` at the end keeps **only** the items that pass.

Sentence: **"the tickers whose last price is above their first."**

NESN went from 104.1 to 102.3, so it is left out. The comprehension also runs over **two lines**: inside brackets, that is allowed (session 3).

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

lined up for the
reader only

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

REMINDER

Indentation (session 3)

Session 3: **indentation is syntax**. After a line ending in `:` (for, if, def), the block must be indented, and the indent decides what belongs to it.

Lines 12–13 are different. Line 13 is indented, but there is **no** `:` and no block: it is the **same statement**, continued inside the open `[`. Here the indent is only for the reader, to line things up.

Line ends with	Next line's indent
<code>:</code>	syntax : starts a block
an open <code>(</code> , <code>[</code> or <code>{</code>	free : a continued statement

prices.py

```
1 # prices.py - nested data, read in one line
2 ledger = {
3     "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4     "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5     "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6 }
7 print(ledger["NESN"]["prices"][-1])
8 print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = {t: d["prices"] * last[t] for t, d in ledger.items()}
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

a new dict

A dict comprehension

```
last = {t: d["prices"][-1] for t, d in ledger.items() }
```

1. loop 2. key: value, for each item

Curly brackets and a **key: value** pair make a **dict**.

Sentence: **"each ticker, with its last price."**

A nested ledger in, a flat dict out: the most common job of a comprehension in data code.

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

Three kinds of comprehension

Brackets	Example	Builds
[...]	[x * 2 for x in shares]	a list
{ k: v ... }	{t: len(t) for t in tickers}	a dict
{ ... }	{d["shares"] for d in ledger.values()}	a set: unique values

The brackets tell you what comes out, before you read anything else.

`sum(x * 2 for x in shares)`, with no brackets of its own, feeds the values straight into `sum`. You will see this often; read it like a list comprehension.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print([p for d in ledger.values()
23       for p in d["prices"]])

```

outer loop

inner loop

```
$ python prices.py
```

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
[185.0, 188.2, 190.5, 104.1, 103.0, 102.3, 505.0, 509.9, 512.0]

```

⚠ WARNING

Two for s: hard to read

Lines 22–23 now hold a comprehension with **two** loops: every price of every position, in one flat list.

The **for** s run **in the order written**, outer first, as if nested:

```

for d in ledger.values():
    for p in d["prices"]:
        keep p

```

This is where comprehensions stop helping. If you need to think about it, a written-out loop is clearer. When an AI writes one, rewrite it as loops before trusting it.

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

+ GOOD TO KNOW

When to use a comprehension

- **Good:** one loop, one simple expression, maybe one `if`. Clearer than four lines.
- **Bad:** two loops, nested conditions, long expressions. Nobody can read it, including you in a month.

A rule of thumb from experienced programmers: if the comprehension does not fit on one or two lines, or you cannot say it in one sentence, **write the loop**.

Both versions run at almost the same speed. Choose the one a reader understands.

PART 3

Lambda and other one-line tricks

Small functions without a name, and
decisions inside an expression.

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]  < a function of p
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

Lambda: a function in one expression

```
def change(p): return (p[-1] - p[0]) / p[0]
```

```
lambda change = lambda p: (p[-1] - p[0]) / p[0]
```

parameters the value it returns

`lambda p: ...` is a small function: parameters before the colon, **one expression** after it, returned automatically.

No `def`, no name, no `return`.

Line 21 gives it a name anyway, `change`, and line 22 calls it like any function: each position's change from first to last price.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 def change(p): return (p[-1] - p[0]) / p[0]  < the def version
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})

```

```
$ python prices.py
```

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}

```

same result

Same function, written with `def`

Line 21 now uses `def`. Output line 7 is unchanged.

A lambda can only hold **one expression**: no loops, no `if` blocks, no several lines. Anything bigger needs `def`.

Style guides (PEP 8, session 2) prefer `def` when you give the function a name, as here. Lambdas are for the place where they shine: **the next slide**.

prices.py

```
1 # prices.py – nested data, read in one line
2 ledger = {
3     "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4     "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5     "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6 }
7 print(ledger["NESN"]["prices"][-1])
8 print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]  ◀ a Greek letter, λ
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

+ GOOD TO KNOW

Why "lambda"?

In the 1930s, the logician **Alonzo Church** invented a mathematics of functions, the **lambda calculus**. He wrote a function with the Greek letter λ (lambda):

1930s · Church

$$\lambda p. (p_2 - p_0) / p_0$$

1958 · Lisp

$$(\text{lambda } (p) \dots)$$

1994 · Python

$$\text{lambda } p: (p[-1] - p[0]) / p[0]$$

The programming language **Lisp** borrowed the word, and Python borrowed it from there. The lambda calculus is also one of the foundations of computer science itself: it describes exactly what can be computed.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger)  # sort the tickers
19
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})

```

```
$ python prices.py
```

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'NESN', 'V00']  # alphabetical
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}

```

Sorting the ledger: alphabetically

We want to put our positions in order. The simplest order needs nothing extra: line 18, `sorted(ledger)`, sorts the **keys**, the tickers, as text: A, N, V.

`sorted` knows how to compare text and numbers by itself. But "alphabetical" is rarely the order we care about. We would rather sort by **shares**, or by **value**.

For that, `sorted` needs to be told **how to turn each ticker into something to compare**. It asks us for a **function**.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 ticker → its shares
17 def n_shares(t): return ledger[t]["shares"]
18 by_value = sorted(ledger, key=n_shares)
19                a function, as input
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})

```

```
$ python prices.py
```

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['V00', 'NESN', 'AAPL'] 2, 4, 10 shares
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}

```

By number of shares: a function as input

Line 17 defines a small function: given a ticker, it returns its number of shares. Line 18 **hands that function to sorted**: `key=n_shares`.

each ticker	"AAPL"	"NESN"	"V00"
	↓ sorted calls n_shares on each one		
compared by	10	4	2
	↓ smallest first		
result	['V00', 'NESN', 'AAPL']		

Some functions take other functions as input.

`key=n_shares` has **no parentheses**: we pass the function itself (session 4), and `sorted` calls it, once per ticker.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17 def value(t): return ledger[t]["shares"] * last[t]
18 by_value = sorted(ledger, key=value, reverse=True)
19
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})

```

how to find a value

passed to sorted

\$ python prices.py

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}

```

largest value first

By value: define how to compute it

Now the order we really want: by the **value** of each position. That number is not stored anywhere, so we **define how to find it** (line 17), then pass that function to `sorted` (line 18):

each ticker	"AAPL"	"NESN"	"V00"
	↓ value(t) = shares × last price		
compared by	1905.0	409.2	1024.0
	↓ largest first (reverse=True)		
result	['AAPL', 'V00', 'NESN']		

Same `sorted`, a different function, a different order. `sorted` does the sorting; **we** decide what it sorts by.

prices.py

```
1 # prices.py – nested data, read in one line
2 ledger = {
3     "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4     "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5     "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6 }
7 print(ledger["NESN"]["prices"][-1])
8 print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

\$ python prices.py

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

The same, with a lambda

value is used **once**, and it is one short expression.

Writing a `def`, and a name, for it is clumsy. The original lines 18–19 write the function **right where it is passed**:

key=value	the named function
key=lambda t: ledger[t]["shares"] * last[t]	the same function, inline

This is what lambdas are for: a short function passed as an argument. Longer, or used in several places: write a `def` and pass its name.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13            if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = max(ledger, key=lambda t: last[t])
19
20 print(by_value, min(ledger, key=lambda t: last[t]))
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})

```

compare by last price

\$ python prices.py

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
V00 NESN
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}

```

highest · lowest

Where lambdas appear: **max** and **min**

max and **min** take the same **key=** as **sorted**: a function saying **what to compare**.

- `max(ledger)` alone would compare the tickers as text: "V00".
- `max(ledger, key=lambda t: last[t])` compares their **last prices**: V00, at 512.0. `min` gives NESN.

Read the lambda as "**for one ticker t: its last price**", and the call as "the ticker with the highest one".

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(last.items(), key=lambda kv: kv[1])
19
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})

```

the second part of a pair

\$ python prices.py

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
[('NESN', 102.3), ('AAPL', 190.5), ('V00', 512.0)]
pairs, cheapest first {'NESN': -0.0173, 'V00': 0.0139}

```

Where lambdas appear: sorting pairs

`last.items()` gives (ticker, price) **pairs** (session 3). Sorted as they are, pairs compare by their **first** part, the ticker.

`key=lambda kv: kv[1]` says: compare by the **second** part, the price. `kv` is just a name for one pair ("key, value"). You will see `lambda x: x[1]` constantly in AI-generated code: it almost always means "by the second element".

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = list(filter(lambda t: last[t] > 150, ledger))
19
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})

```

True or False, per ticker

\$ python prices.py

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}

```

kept: True

Where lambdas appear: **filter**

`filter(f, items)` keeps the items for which `f` returns `True`. Here the function is a lambda that asks one question per ticker: **is its last price above 150?**

`filter` gives back its results one by one; `list(...)` collects them into a list.

It does the same as a comprehension with an `if` (Part 2): `[t for t in ledger if last[t] > 150]`, usually easier to read.

Once the prices are a numpy array or a pandas column, there is a nicer way still: a **boolean mask**, `prices[prices > 150]`. We will see it **later today** (Parts 4–5). A plain dict has no masks.

```

tables.py
1 # tables.py – pandas: a table with names
2 import pandas as pd
3
4 df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                       "sector": ["tech", "tech", "food", "fund"],
6                       "shares": [10, 3, 4, 2],
7                       "price": [190.5, 410.0, 102.3, 512.0]})
8 df["value"] = df["shares"] * df["price"]
9 print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df.groupby("sector")["value"].sum())

```

a function for one value

```
$ python tables.py
```

```

  ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   V00   fund       2   512.0  1024.0
(4, 5) 4568.2
NESN 190.5
  ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0
sector
food      409.2
fund     1024.0
tech     3135.0
Name: value, dtype: float64

```

Where lambdas appear: pandas (a preview)

A glimpse ahead: in pandas (Part 5), `.apply` takes a function and calls it **on every value of a column**. Lines 15–16 pass it a lambda.

Ignore the pandas details for now; read only the lambda: **"for one value `v`: "large" if above 1500, otherwise "small".** Part 5 shows what `.apply` does with it.

Everywhere, the pattern is the same: **a function that expects a function**, and a short lambda written right there.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: "up" if p[-1] > p[0] else "down"
22 print({t: change(d["prices"])
23       for t, d in ledger.items()})

```

```
$ python prices.py
```

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 'up', 'NESN': 'down', 'V00': 'up'}

```

one word per position

if ... else inside an expression

Line 21 now returns "up" or "down". The decision sits **inside one expression**:

read it as **"up" if p[-1] > p[0] else "down"**
 the value · when this is True · otherwise this value

This is a **conditional expression**. It gives a value, so it fits where `if` blocks cannot: in a lambda, a comprehension, a function argument. Read it middle first: the condition, then which side wins.

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

Reading compact code: a checklist

You see	Ask
a[x] [y] [z]	What type do I have after each bracket?
[... for ... in ...]	What does it go round? What does it keep? Which brackets?
... if ...] at the end	Which items are left out?
lambda x: ...	What does it compute for one x? Who calls it?
a if cond else b	Which side wins, and when?
key=	Sorted by what?

Say each line **in one English sentence** before you move on. If you cannot, slow down: rewrite it as a loop, or as a `def`.

PART 4

numpy, properly

Grids of numbers, and operations on all of them at once.

arrays.py

```
1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
```

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

What numpy adds to Python

Plain Python

- a list of numbers is a list of **separate objects**
- arithmetic on a list? `[10, 4, 2] * 2` repeats it
- one number at a time, in a **loop** you write
- a table of numbers = a list of lists, awkward to slice

With numpy

- one **block of numbers**, all of one type
- arithmetic on **whole arrays**: `shares * prices`
- no loop to write, and **tens of times faster**
- real **grids**: `prices[:, -1]`, `axis=`, broadcasting

In short: **numpy turns Python into a language for computing with large amounts of numbers.** Almost every scientific and data library, pandas included, is built on top of it.

arrays.py

```
1 # arrays.py – numpy: a grid of numbers of one type
2 import numpy as np
3
4 prices = np.array([[185.0, 188.2, 190.5],
5                   [104.1, 103.0, 102.3],
6                   [505.0, 509.9, 512.0]])
7 shares = np.array([10, 4, 2])
8 print(prices.shape, prices.dtype, prices.ndim)
9 print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

\$ python arrays.py

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.  409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

An array with rows and columns

A new script on the left. Lines 4–6 turn a **list of lists** into a two-dimensional array: one row per position, one column per day.

	day 0	day 1	day 2
AAPL 0	185	188.2	190.5
NESN 1	104.1	103	102.3
VOO 2	505	509.9	512

rows: positions · columns: days

- `shape` is `(3, 3)`: 3 rows, 3 columns.
- `dtype` is `float64`: every element has the **same** type (session 3).
- `ndim` is `2`: two directions to index in.

```

1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, back to a list of lists])
8  print(type(prices), type(prices.tolist()))
9  print(prices[0] * 2, prices.tolist()[0] * 2)
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())

```

```
$ python arrays.py
```

```

<class 'numpy.ndarray'> <class 'list'> two different types
[370.  376.4 381. ] [185.0, 188.2, 190.5, 185.0, 188.2, 190.5]
[[188.2 190.5]
 [103.  102.3]]
list: * repeats
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4

```

+ GOOD TO KNOW

Python has no arrays of its own

Python itself has **no grid-of-numbers type**. (Its standard library has a small `array` module for flat lists of numbers; almost nobody uses it.) Grids come from numpy.

An array is **written and printed like a list of lists**, but that is only its look. On the left, line 8 compares its type with a real list of lists: `numpy.ndarray` against `list`.

Different type, different tools (session 4): line 9 doubles one row both ways. The array **multiplies**; the list **repeats**. `.shape`, `axis=` and masks exist only for the array; `.append` only for the list.

```

1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, shares.shape)
9  print(shares[:, None].shape, shares[None, :].shape)
10 print(shares[:, None])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())

```

```
$ python arrays.py
```

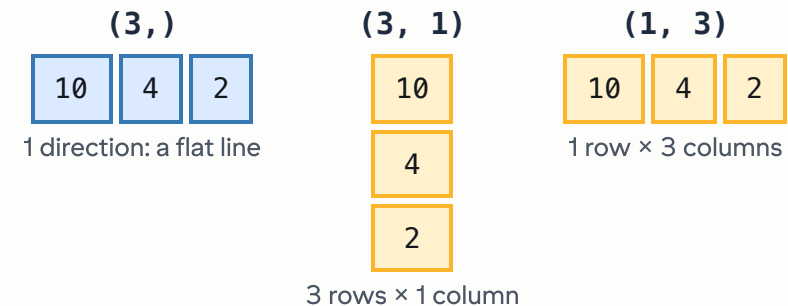
```

(3, 3) (3,)  ← one number: one direction
(3, 1) (1, 3) ← 3 rows, 1 column
[[10]
 [ 4]
 [ 2]]
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4

```

What **shape** means

shape lists **how many elements lie along each direction**. One number per direction, in a tuple:



Same three numbers, three **different shapes**. **(3,)** has **no rows and no columns**, just one direction; the comma only marks it as a tuple. **(3, 1)** is a real **column**: `shares[:, None]` adds the second direction.

Shapes decide how arrays combine (broadcasting, later). When numpy surprises you, print the `.shape` first.

```

1 # arrays.py – numpy: a grid of numbers of one type
2 import numpy as np
3
4 prices = np.array([[185.0, 188.2, 190.5],
5                   [104.1, 103.0, 102.3],
6                   [505.0, 509.9, 512.0]])
7 shares = np.array([10, 4, 2])
8 row = shares[None, :] < the same 3 numbers, as a row
9 print(shares.ndim, row.ndim, len(shares), len(row))
10 print(shares[0], row[0], shares.T.shape, row.T.shape)
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())

```

```
$ python arrays.py
```

```

1 2 3 1
10 [10  4  2] (3,) (3, 1) < [0]: a number · a whole row
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.   509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4

```

+ GOOD TO KNOW**(3,) or (1, 3) ?**

They print almost alike, `[10 4 2]` and `[[10 4 2]]`, but they are not the same thing:

	shares, (3,)	row, (1, 3)
directions (ndim)	1	2
len(...)	3	1: one row
[0] gives	10, a number	[10 4 2], the whole row
.T (transpose)	(3,) : nothing changes	(3, 1) : a column

Rule of thumb: `(3,)` is **"3 numbers"**; `(1, 3)` and `(3, 1)` are **a table one row or one column wide**. In broadcasting, a `(3,)` acts like a row, which is why a column needs `[:, None]`.

arrays.py

```
1 # arrays.py – numpy: a grid of numbers of one type
2 import numpy as np
3
4 prices = np.array([[185.0, 188.2, 190.5],
5                   [104.1, 103.0, 102.3],
6                   [505.0, 509.9, 512.0]])
7 shares = np.array([10, 4, 2])
8 print(prices.shape, prices.dtype, prices.ndim)
9 print(prices[1, 2], prices[1], prices[:, -1]) # row 1, column 2
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
```

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ] # one number
[[188.2 190.5]
 [103.  102.3]]
[1905.  409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

Indexing: [row, column]

One comma, two positions, both from 0:

	day 0	day 1	day 2
AAPL 0	185	188.2	190.5
NESN 1	104.1	103	102.3
VOO 2	505	509.9	512

prices[1, 2]: row 1 (NESN), column 2
(day 2)

A list of lists would need `prices[1][2]`. numpy takes both positions in **one** pair of brackets, separated by a comma.

arrays.py

```
1 # arrays.py – numpy: a grid of numbers of one type
2 import numpy as np
3
4 prices = np.array([[185.0, 188.2, 190.5],
5                   [104.1, 103.0, 102.3],
6                   [505.0, 509.9, 512.0]])
7 shares = np.array([10, 4, 2])
8 print(prices.shape, prices.dtype, prices.ndim)
9 print(prices[1, 2], prices[1], prices[:, -1]) # a row
10 print(prices[:2, 1:]) # a column
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.  409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

Rows and columns

	day 0	day 1	day 2		day 0	day 1	day 2
AAPL	185	188.2	190.5	AAPL	185	188.2	190.5
NESN	104.1	103	102.3	NESN	104.1	103	102.3
VOO	505	509.9	512	VOO	505	509.9	512

prices[1]: a whole row

prices[:, -1]: a whole column

: alone means "all of this dimension". `prices[:, -1]` reads "every row, last column": each position's latest price.

```

1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])  # rows 0–1, columns 1–2
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())

```

```
$ python arrays.py
```

```

(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.  409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4

```

Slicing in two directions

Each position in the brackets takes a **slice** (session 3), separately:

	day 0	day 1	day 2
AAPL 0	185	188.2	190.5
NESN 1	104.1	103	102.3
VOO 2	505	509.9	512

prices[:2, 1:]: rows up to 2, columns from 1

The result is a smaller 2×2 array: AAPL and NESN, days 1 and 2.

Read it dimension by dimension: **rows first, then columns**. Everything you know about `start:stop:step` works in each slot.

arrays.py

```
1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

Indexing cheat sheet

You write	You get	Shape
prices[1, 2]	one number	—
prices[1] or prices[1, :]	row 1	(3,)
prices[:, 0]	column 0	(3,)
prices[:2, 1:]	a block	(2, 2)
prices[::-1]	rows in reverse order	(3, 3)
prices[:, [0, 2]]	columns 0 and 2 (a list of positions)	(3, 2)

Count the **scalars** (single numbers) in the brackets:
each one removes a dimension. `:` and slices keep it.

arrays.py

```
1 # arrays.py – numpy: a grid of numbers of one type
2 import numpy as np
3
4 prices = np.array([[185.0, 188.2, 190.5],
5                   [104.1, 103.0, 102.3],
6                   [505.0, 509.9, 512.0]])
7 shares = np.array([10, 4, 2])
8 print(prices.shape, prices.dtype, prices.ndim)
9 print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1] < 3 multiplications, no loop
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

\$ python arrays.py

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.   409.2 1024. ] 3338.2 < the total, again
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

Element by element (session 3), now for real

Line 12 multiplies the shares by the last column: **three multiplications, no loop**. Line 13 adds them up: 3338.2, session 3's total.

The rule from session 3: arithmetic on arrays works **element by element**, position 0 with position 0, and so on.

This is called **vectorised** code: you write the operation once, for the whole array.

arrays.py

```
1 # arrays.py – numpy: a grid of numbers of one type
2 import numpy as np
3
4 prices = np.array([[185.0, 188.2, 190.5],
5                   [104.1, 103.0, 102.3],
6                   [505.0, 509.9, 512.0]])
7 shares = np.array([10, 4, 2])
8 print(prices.shape, prices.dtype, prices.ndim)
9 print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.  409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

axis: which direction to collapse

	day 0	day 1	day 2		day 0	day 1	day 2
AAPL	185	188.2	190.5	AAPL	185	188.2	190.5
NESN	104.1	103	102.3	NESN	104.1	103	102.3
VOO	505	509.9	512	VOO	505	509.9	512

axis=1 → one mean per row: 187.9 · 103.13 · 508.97

axis=0 → one max per column: 505 · 509.9 · 512

axis=1 squeezes the **columns** away: one result per row (position). **axis=0** squeezes the **rows** away: one result per column (day). No **axis**: one number for everything.

```

1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=0).round(2), prices.max(axis=1))
15 history = shares[0] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())

```

```

$ python arrays.py
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.  409.2 1024. ] 3338.2
[264.7  267.03 268.27] [190.5 104.1 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4

```

⚠ WARNING

The wrong axis runs fine

Line 14 now swaps the axes. The output is **three numbers again**, just different ones: the mean of each **day**, and the max of each **position**.

No error: both are valid questions. Only you know which one you meant. When you see `axis=` in AI-generated code, **say the question out loud**: "the mean of each row, or of each column?"

arrays.py

```
1 # arrays.py – numpy: a grid of numbers of one type
2 import numpy as np
3
4 prices = np.array([[185.0, 188.2, 190.5],
5                   [104.1, 103.0, 102.3],
6                   [505.0, 509.9, 512.0]])
7 shares = np.array([10, 4, 2])
8 print(prices.shape, prices.dtype, prices.ndim)
9 print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices <math>(3,) \rightarrow (3, 1)</math>
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
```

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

The idiom `[:, None]`: turn a line into a column

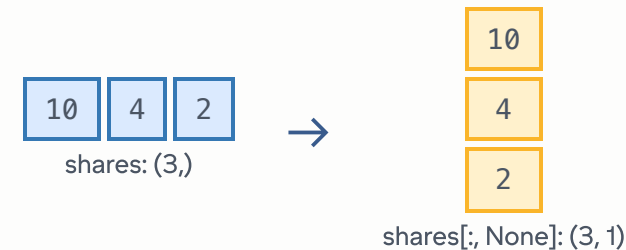
`shares[:, None]`

`:` first slot

keep all the elements, along the existing direction

`None` second slot

add a **new direction** of length 1



No numbers change: only the **shape**. The reverse, `shares[None, :]`, gives a row `(1, 3)`. You will also see it written `np.newaxis`, which is just another name for `None`. Now broadcasting knows to stretch the column **across the days**: next slide.

arrays.py

```
1 # arrays.py – numpy: a grid of numbers of one type
2 import numpy as np
3
4 prices = np.array([[185.0, 188.2, 190.5],
5                   [104.1, 103.0, 102.3],
6                   [505.0, 509.9, 512.0]])
7 shares = np.array([10, 4, 2])
8 print(prices.shape, prices.dtype, prices.ndim)
9 print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices  # a column times a grid
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
```

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]  # the portfolio, each day
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

Broadcasting: stretching to fit

`shares[:, None]` turns the 3 shares into a **column**. Multiplied by the 3×3 grid, the column is **stretched** across the days:

10	×	185.0	188.2	190.5	=	1850	1882	1905
4		104.1	103.0	102.3		416.4	412	409.2
2		505.0	509.9	512.0		1010	1019.8	1024

numpy does this **automatically**, whenever the shapes can be made to fit: no copies, no loop. Every position's value on every day; line 16 adds each **column**: the portfolio on days 0, 1, 2.

```

1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 history = shares * prices  < [:, None] forgotten
15 print(shares.shape, prices.shape, history.shape)
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())

```

```
$ python arrays.py
```

```

(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.  409.2 1024. ] 3338.2
(3,) (3, 3) (3, 3)  < the shapes fit
[7941.  3204.4 1609.6]  < no error, nonsense
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4

```

⚠ WARNING

Wrong broadcasting: an invisible error

Line 14 has lost `[:, None]`. Line 15 prints the shapes: they still **fit**, so numpy multiplies **without a word**:

intended	<u>(3, 1)</u>	× (3, 3)	a column: stretched across the days
written	<u>(3,)</u>	× (3, 3)	read as a row (1, 3): stretched down the positions

Now each **day** is multiplied by 10, 4 or 2. The portfolio "falls" from 7941 to 1610; the truth is 3276 to 3338.

The **result's** shape, (3, 3), looks right either way. Compare the **inputs'** shapes with what you meant, and check one number by hand: AAPL on day 0 should be $10 \times 185 = 1850$.

arrays.py

```
1 # arrays.py – numpy: a grid of numbers of one type
2 import numpy as np
3
4 prices = np.array([[185.0, 188.2, 190.5],
5                   [104.1, 103.0, 102.3],
6                   [505.0, 509.9, 512.0]])
7 shares = np.array([10, 4, 2])
8 print(prices.shape, prices.dtype, prices.ndim)
9 print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history, "tomorrow / today")
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
```

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.  409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

Daily returns: two shifted slices

	day 0	day 1	day 2		day 0	day 1	day 2
AAPL	185	188.2	190.5	AAPL	185	188.2	190.5
NESN	104.1	103	102.3	NESN	104.1	103	102.3
VOO	505	509.9	512	VOO	505	509.9	512

prices[:, 1:]: days 1–2

prices[:, :-1]: days 0–1

Divide the two shifted views, element by element: each day's price over the day before. Minus 1 gives the **return**: NESN's are negative both days. One line replaces two nested loops.

arrays.py

```
1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

a mask, then a filter count the Trues

```
$ python arrays.py
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

Boolean masks (session 3), with a count

`values > 1000` gives `[True, False, True]`. Used **inside** the brackets, it keeps the `True` positions: AAPL's and VOO's values.

`(returns > 0).sum()` counts the positive returns: `True` counts as 1 (session 2's coercion). **4** of the 6 daily returns were positive.

Masks are the numpy way of writing session 3's "collect" and "count" patterns.

```

1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14
15 history = shares[:, None] * prices
16
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18
19 print(values[(values > 1000) and (values < 2000)])

```

Python's and: one True/False

```

$ python arrays.py
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.  409.2 1024. ] 3338.2
Traceback (most recent call last):
  File "arrays.py", line 19, in <module>
    print(values[(values > 1000) and (values < 2000)])
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: The truth value of an array with more than one element
is ambiguous. Use a.any() or a.all()

```

⚠ WARNING

Combine masks with **&**, not **and**

Line 19 now combines two masks with **and**. numpy refuses: **and** needs **one** True or False, and an array holds many.

Python, one value	numpy, element by element
and	&
or	
not	~

Correct: `values[(values > 1000) & (values < 2000)]`.
 The **parentheses are required**: **&** binds tighter than **>**.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13            if d["prices"][-1] > d["prices"][0]]
14 print(a copy of the outer dict only)
15 c = ledger.copy()
16 c["NESN"]["shares"] = 0  < changes an inner dict
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value, ledger["NESN"]["shares"])
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})

```

```
$ python prices.py
```

```

102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
['AAPL', 'V00', 'NESN'] 0
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}

```

⚠ WARNING**.copy() copies one level only**

Line 15 makes a copy `c`, and line 16 changes `c`. Yet line 20 shows that `ledger`'s NESN now has 0 shares.

`.copy()` makes a new **outer** dict, but the inner dicts are **shared**: two names, one object (sessions 3 and 4), one level down.

For a full, independent copy: `import copy` and `copy.deepcopy(ledger)`. numpy has the same trap, in another form: next slide.

```

1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 row = prices[1]  ← a view, not a copy
11 row[0] = 0.0
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())

```

```

$ python arrays.py
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[1905.  409.2 1024. ] 3338.2
[187.9  68.43 508.97] [505.  509.9 512. ] ← NESN's mean changed
[2860.  3313.8 3338.2]
[[ 0.0173  0.0122]
 [    inf -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 5
arrays.py:17: RuntimeWarning: divide by zero encountered in divide
returns = prices[:, 1:] / prices[:, :-1] - 1

```

⚠ WARNING

A slice of an array is a view

Line 10 takes row 1, and line 11 changes the copy... except it is not a copy. `prices` itself changed: NESN's mean drops to **68.43**, and a return becomes `inf` (division by zero).

List slices are copies (session 3). **numpy slices are views**: another way of looking at the same numbers. Session 3's slicing warning, come true.

For an independent copy: `prices[1].copy()`.

arrays.py

```
1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
```

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

numpy functions you will meet

Make	Summarise	Transform
np.array(list)	.sum() .mean() .std()	np.log(a) np.sqrt(a)
np.zeros(3) np.ones((2, 3))	.min() .max() .argmax()	a.round(2) np.abs(a)
np.arange(0, 10, 2)	np.cumsum(a)	a.reshape(3, 1) a.T
np.linspace(0, 1, 5)	np.percentile(a, 90)	np.where(cond, x, y)

Most take `axis=`. Most work element by element.
When a name is new: `help(np.cumsum)`, or hover in VS Code.

arrays.py

```
1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
```

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

+ GOOD TO KNOW

Why numpy is fast

A Python list holds **references** to separate number objects scattered in memory. A numpy array holds the numbers themselves, **side by side**, all of one type.

That lets numpy run its loops in fast compiled code (C) instead of Python, often **10 to 100 times faster** on large arrays.

This is session 1's machine, used well: memory laid out in a row, processors doing the same arithmetic on many numbers. pandas, scikit-learn and JAX are built on the same idea.

PART 5

pandas, properly

Tables with named columns: the shape of almost all real data.

```

tables.py
1 # tables.py – pandas: a table with names
2 import pandas as pd
3
4 df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8 df["value"] = df["shares"] * df["price"]
9 print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df.groupby("sector")["value"].sum())

```

```
$ python tables.py
```

```

  ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   V00   fund       2   512.0  1024.0

```

```
(4, 5) 4568.2
```

```
NESN 190.5
```

```

  ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0

```

```

sector
food      409.2
fund     1024.0
tech     3135.0

```

```
Name: value, dtype: float64
```

What pandas adds to Python

Plain Python (or numpy)

- a table = a list of dicts, or a grid of **anonymous** numbers
- columns of different types? numpy wants one type
- "total value per sector": you write the loop, the dict, the sums
- reading a CSV or Excel file: many lines of your own code

With pandas

- a **table with names**: columns and row labels
- each column its **own type**: text, numbers, dates
- `df.groupby("sector")["value"].sum()` : **one line**
- `pd.read_csv("ledger.csv")` : **one line**, and back with `.to_csv`

In short: **pandas is the spreadsheet of Python**, programmable and reproducible, and fast because it is built on numpy. Part 2 of the course lives in it.

```

tables.py
1 # tables.py – pandas: a table with names
2 import pandas as pd
3
4 df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8 df["value"] = df["shares"] * df["price"]
9 print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df.groupby("sector")["value"].sum())

```

```
$ python tables.py
```

```

  ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   V00   fund       2   512.0  1024.0

```

column names

```
(4, 5) 4568.2
```

```
NESN 190.5
```

```

  ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0

```

```

sector
food      409.2
fund     1024.0
tech     3135.0

```

```
Name: value, dtype: float64
```

A DataFrame: rows, columns, an index

	ticker	sector	shares	price	value
0	AAPL	tech	10	190.5	1905.0
1	MSFT	tech	3	410.0	1230.0
2	NESN	food	4	102.3	409.2
3	V00	fund	2	512.0	1024.0

column names

the index: row labels

Lines 4–7 build it from a **dict of lists**: each key becomes a column. Line 8 adds a column computed from two others, element by element, like numpy.

```

tables.py
1 # tables.py – pandas: a table with names
2 import pandas as pd
3
4 df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8 df["value"] = df["shares"] * df["price"]
9 print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, df["value"].sum()])
12
13 big = df[df["value"] > 1000]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df.groupby("sector")["value"].sum())

```

```

$ python tables.py
  ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   V00   fund       2   512.0  1024.0
(4, 5) 4568.2 the column's sum
NESN 190.5
  ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0
sector
food      409.2
fund     1024.0
tech     3135.0
Name: value, dtype: float64

```

A column is a Series (session 4)

`df["value"]` picks one column, by name, like a dict key. The result is a **Series**: a one-dimensional array with the index attached.

A Series has the numpy methods: `.sum()`, `.mean()`, `.max()`. `df["value"].sum()` is **4568.2**: session 3's ledger plus MSFT.

Two brackets pick **several** columns: `df[["ticker", "value"]]` is a smaller DataFrame. Line 14 does exactly this.

```

tables.py
1 # tables.py - pandas: a table with names
2 import pandas as pd
3
4 df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8 df["value"] = df["shares"] * df["price"]
9 print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df.groupby("sector")["value"].sum())

```

```

$ python tables.py
   ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   V00   fund       2   512.0  1024.0
(4, 5) 4568.2

```

NESN 190.5 ← row 2 ticker · row 0 column 3

```

   ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0
sector
food      409.2
fund     1024.0
tech     3135.0
Name: value, dtype: float64

```

.loc and .iloc: labels or positions

You write	Looks up by	Gives
<code>df.loc[2, "ticker"]</code>	labels: index label 2, column named "ticker"	NESN
<code>df.iloc[0, 3]</code>	positions: row 0, column 3 (from 0)	190.5

Both use numpy's `[row, column]` idea. `.loc` reads the **names** printed on the table; `.iloc` counts **positions**, like a list.

Right now the index labels are 0, 1, 2, 3, so the two look alike. They stop looking alike after the next slide.

```

tables.py
1 # tables.py - pandas: a table with names
2 import pandas as pd
3
4 df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8 df["value"] = df["shares"] * df["price"]
9 print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000] < keep rows where True
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df.groupby("sector")["value"].sum())

```

\$ python tables.py

```

  ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   V00   fund       2   512.0  1024.0

```

(4, 5) 4568.2

NESN 190.5

```

  ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0

```

NESN (label 2) is gone

```

sector
food    409.2
fund    1024.0
tech    3135.0

```

Name: value, dtype: float64

Filtering rows with a mask

`df["value"] > 1000` is a Series of True / False, one per row. `df[mask]` keeps the True rows (numpy's masks, again):

0	AAPL	1905.0	→ True	kept
1	MSFT	1230.0	→ True	kept
2	NESN	409.2	→ False	dropped
3	V00	1024.0	→ True	kept

The rows **keep their labels**: 0, 1, 3. Label 2 is simply missing now.

```

1  # tables.py – pandas: a table with names
2  import pandas as pd
3
4  df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "VOO"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8  df["value"] = df["shares"] * df["price"]
9  print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000]
14 print(big.loc[3, "ticker"], big.iloc[2, 0])
15 df["size"] = df["value"].apply(lambda v: "large" if v > 1500 else "small")
16
17 print(df.groupby("sector")["value"].sum())

```

\$ python tables.py

```

   ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   VOO   fund       2   512.0  1024.0
(4, 5) 4568.2
NESN 190.5
VOO VOO
sector
food      409.2
fund     1024.0
tech     3135.0
Name: value, dtype: float64

```

⚠ WARNING

After a filter, labels and positions differ

In `big`, VOO has **label 3** but sits at **position 2**. Line 14 now reaches it both ways: `big.loc[3, ...]` and `big.iloc[2, ...]`.

`big.loc[2, ...]` would now fail: no row is labelled 2.

`big.iloc[3, ...]` would fail too: there are only 3 rows.

Know which one you mean. Most bugs with pandas indexing come from mixing the two. In AI-generated code, check whether it uses `.loc` or `.iloc`, and why.

```

1  # tables.py – pandas: a table with names
2  import pandas as pd
3
4  df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8  df["value"] = df["shares"] * df["price"]
9
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000 and df["sector"] == "tech"]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df.groupby("sector")["value"].sum())

```

```

$ python tables.py
(4, 5) 4568.2
NESN 190.5
Traceback (most recent call last):
  File "tables.py", line 13, in <module>
    big = df[df["value"] > 1000 and df["sector"] == "tech"]
    ~~~~~^~~~~~
  File ".../site-packages/pandas/core/generic.py", line 1501, in
__bool__
    raise ValueError(
ValueError: The truth value of a Series is ambiguous. Use a.empty,
a.bool(), a.item(), a.any() or a.all().

```

⚠ WARNING

Two conditions: **&**, with parentheses

Line 13 now combines two conditions with **and**, and pandas refuses, with the same message as numpy: a Series is not **one** True or False.

The traceback runs **deep into pandas' own files** before the last line. Look for your own line first, then read the last line.

Correct: `df[(df["value"] > 1000) & (df["sector"] == "tech")]`. Same rule as numpy: **&**, **|**, **~**, and parentheses around each condition.

tables.py **CHANGED ON THIS SLIDE**

```
1 # tables.py – pandas: a table with names
2 import pandas as pd
3
4 df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                     "sector": ["tech", "tech", "food", "fund"],
6                     "shares": [10, 3, 4, 2],
7                     "price": [190.5, 410.0, 102.3, 512.0]})
8 df["value"] = df["shares"] * df["price"]
9 print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df[["ticker", "value", "size"]])
```

Part 3's lambda

```
$ python tables.py
```

```
   ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   V00   fund       2   512.0  1024.0
```

```
(4, 5) 4568.2
```

```
NESN 190.5
```

```
   ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0
```

```
   ticker  value  size
0  AAPL  1905.0  large
1  MSFT  1230.0  small
2  NESN   409.2  small
3   V00  1024.0  small
```

.apply: a function on every value

Lines 15–16 add a column `size`: the lambda from Part 3, with a conditional expression, called once **for each value** of the column. Line 17 now shows the result.

Read it the checklist way: **"for each value v: "large" if above 1500, otherwise "small" ."**

`.apply` is flexible but slow on big tables. When a numpy-style operation exists, prefer it:

`np.where(df["value"] > 1500, "large", "small")` does the same, vectorised.

```

tables.py
1 # tables.py – pandas: a table with names
2 import pandas as pd
3
4 df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8 df["value"] = df["shares"] * df["price"]
9 print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "big" if v > 1500 else "small")
17 print(df.groupby("sector")["value"].sum())

```

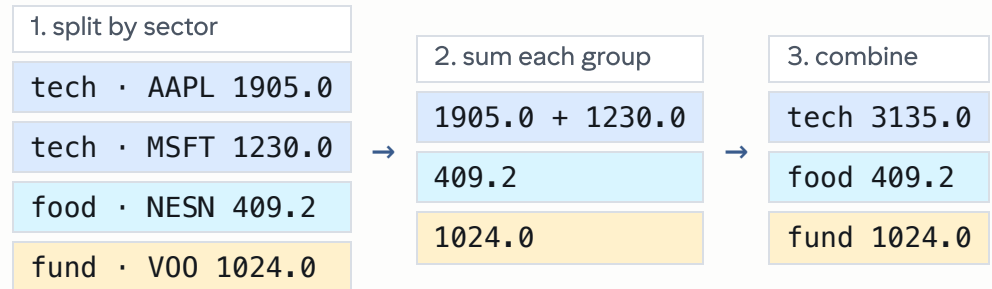
```

$ python tables.py
   ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   V00   fund       2   512.0  1024.0
(4, 5) 4568.2
NESN 190.5
   ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0
sector
food    409.2
fund   1024.0
tech   3135.0
Name: value, dtype: float64

```

1905 + 1230

groupby: split, apply, combine



Read line 17 as a sentence: **"group by sector, take the value column, sum it."** One line, and the question of Part 4 in session 1 ("where did the money go?") has an answer by sector.

```

1  # tables.py – pandas: a table with names
2  import pandas as pd
3
4  df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8  df["value"] = df["shares"] * df["price"]
9  print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df.sort_values("value", ascending=False).head(2))

```

1. sort

2. first two rows

```
$ python tables.py
```

```

  ticker sector  shares  price  value
0  AAPL   tech      10   190.5  1905.0
1  MSFT   tech       3   410.0  1230.0
2  NESN   food       4   102.3   409.2
3   V00   fund       2   512.0  1024.0

```

```
(4, 5) 4568.2
```

```
NESN 190.5
```

```

  ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0

```

```

  ticker sector  shares  price  value  size
0  AAPL   tech      10   190.5  1905.0  large
1  MSFT   tech       3   410.0  1230.0  small

```

Method chains: read left to right

Line 17 now sorts the table by value, largest first, and keeps the first two rows (session 4's chain).

Long pandas chains are common, especially from AI assistants:

```

(df[df["value"] > 500]
 .groupby("sector")["value"]
 .sum()
 .sort_values(ascending=False))

```

Read each line as one step, and **ask what type** comes out of each: DataFrame, grouped data, Series, Series. Outer parentheses let the chain span several lines.

```
$ python tables.py
  ticker sector  shares  price  value
0  AAPL   tech     10  190.5  1905.0
1  MSFT   tech      3  410.0  1230.0
2  NESN   food      4  102.3   409.2
3   VOO   fund      2  512.0  1024.0
(4, 5) 4568.2
NESN 190.5
  ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   VOO  1024.0
sector
food    409.2
fund   1024.0
tech   3135.0
Name: value, dtype: float64
```

Look	Select	Change	Summarize
<code>df.head()</code>	<code>df["col"]</code>	<code>df["new"] = ...</code>	<code>.sum()</code> <code>.mean()</code>
<code>df.shape</code>	<code>df[mask]</code>	<code>.sort_values("col")</code>	<code>.groupby("cat").size()</code>
<code>df.describe()</code>	<code>.loc[...]</code> <code>.iloc[...]</code>	<code>.apply(f)</code>	<code>.value_counts()</code>
<code>df.info()</code>	<code>df[["a", "b"]]</code>	<code>.rename(columns={...})</code>	<code>.agg(["min", "max"])</code>

Data Science and Advanced Programming · UNIL · 2026 74

```

tables.py
1 # tables.py – pandas: a table with names
2 import pandas as pd
3
4 df = pd.DataFrame({"ticker": ["AAPL", "MSFT", "NESN", "V00"],
5                      "sector": ["tech", "tech", "food", "fund"],
6                      "shares": [10, 3, 4, 2],
7                      "price": [190.5, 410.0, 102.3, 512.0]})
8 df["value"] = df["shares"] * df["price"]
9 print(df)
10 print(df.shape, df["value"].sum())
11 print(df.loc[2, "ticker"], df.iloc[0, 3])
12
13 big = df[df["value"] > 1000]
14 print(big[["ticker", "value"]])
15 df["size"] = df["value"].apply(
16     lambda v: "large" if v > 1500 else "small")
17 print(df.groupby("sector")["value"].sum())

```

```

$ python tables.py
  ticker sector  shares  price  value
0  AAPL   tech      10  190.5  1905.0
1  MSFT   tech       3  410.0  1230.0
2  NESN   food       4  102.3   409.2
3   V00   fund       2  512.0  1024.0
(4, 5) 4568.2
NESN 190.5
  ticker  value
0  AAPL  1905.0
1  MSFT  1230.0
3   V00  1024.0
sector
food      409.2
fund     1024.0
tech     3135.0
Name: value, dtype: float64

```

Three ways to hold the same ledger

	Nested dicts (Part 1)	numpy (Part 4)	pandas (Part 5)
Holds	anything, any shape	numbers of one type	columns, each of one type
Look up by	keys and positions	positions	labels or positions
Fast on big data	no	yes	yes
Best for	irregular data, JSON	grids, simulations	tables, real datasets

Part 2 of the course lives mostly in pandas, with numpy underneath. Nested dicts are how data often **arrives**.

```

1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][0] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})

```

a classmate's version

\$ python prices.py

```

102.3
[188.2, 190.5]
{'AAPL': 185.0, 'NESN': 104.1, 'V00': 505.0}
[1850.0, 416.4, 1010.0] 3276.4 is this the total?
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}

```

Spot the lie

A classmate writes line 10 from scratch. It runs. `last` is a dict of prices, the values and their total look like money.

Is it right? Read it with the checklist: what does it go round, and **what does it keep**?

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13           if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23       for t, d in ledger.items()})
```

the last price

The lie, explained

`d["prices"][0]` is the **first** price of each position, not the last: the dict is called `last`, but it holds the prices from **two days ago**. One character, `0` instead of `-1`, and every line after it is wrong.

Failure kind 3: it runs and lies. **Catch it** by saying the line in one sentence: "each ticker with its **first** price". The name promised "last". Or check against a number you trust: session 3's total, 3338.2.

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

the right total

prices.py

```
1  # prices.py – nested data, read in one line
2  ledger = {
3      "AAPL": {"shares": 10, "prices": [185.0, 188.2, 190.5]},
4      "NESN": {"shares": 4, "prices": [104.1, 103.0, 102.3]},
5      "V00": {"shares": 2, "prices": [505.0, 509.9, 512.0]},
6  }
7  print(ledger["NESN"]["prices"][-1])
8  print(ledger["AAPL"]["prices"][1:])
9
10 last = {t: d["prices"][-1] for t, d in ledger.items()}
11 values = [d["shares"] * last[t] for t, d in ledger.items()]
12 rising = [t for t, d in ledger.items()
13            if d["prices"][-1] > d["prices"][0]]
14 print(last)
15 print(values, sum(values))
16 print(rising)
17
18 by_value = sorted(ledger, reverse=True,
19                   key=lambda t: ledger[t]["shares"] * last[t])
20 print(by_value)
21 change = lambda p: (p[-1] - p[0]) / p[0]
22 print({t: round(change(d["prices"]), 4)
23        for t, d in ledger.items()})
```

```
$ python prices.py
```

```
102.3
[188.2, 190.5]
{'AAPL': 190.5, 'NESN': 102.3, 'V00': 512.0}
[1905.0, 409.2, 1024.0] 3338.2
['AAPL', 'V00']
['AAPL', 'V00', 'NESN']
{'AAPL': 0.0297, 'NESN': -0.0173, 'V00': 0.0139}
```

Read it again

The script from the start. Every line is now readable:

- lines 2–6: a dict of dicts of lists, a **tree**;
- lines 7–8: chains of look-ups, ending in an index and a slice;
- lines 10–13: a dict comprehension, a list comprehension, and one with a filter;
- lines 18–19: `sorted` with a `key=` lambda;
- lines 21–23: a named lambda, used inside a dict comprehension.

arrays.py

```
1  # arrays.py – numpy: a grid of numbers of one type
2  import numpy as np
3
4  prices = np.array([[185.0, 188.2, 190.5],
5                    [104.1, 103.0, 102.3],
6                    [505.0, 509.9, 512.0]])
7  shares = np.array([10, 4, 2])
8  print(prices.shape, prices.dtype, prices.ndim)
9  print(prices[1, 2], prices[1], prices[:, -1])
10 print(prices[:2, 1:])
11
12 values = shares * prices[:, -1]
13 print(values, values.sum())
14 print(prices.mean(axis=1).round(2), prices.max(axis=0))
15 history = shares[:, None] * prices
16 print(history.sum(axis=0))
17 returns = prices[:, 1:] / prices[:, :-1] - 1
18 print(returns.round(4))
19 print(values[values > 1000], (returns > 0).sum())
```

```
$ python arrays.py
```

```
(3, 3) float64 2
102.3 [104.1 103.  102.3] [190.5 102.3 512. ]
[[188.2 190.5]
 [103.  102.3]]
[1905.   409.2 1024. ] 3338.2
[187.9  103.13 508.97] [505.  509.9 512. ]
[3276.4 3313.8 3338.2]
[[ 0.0173  0.0122]
 [-0.0106 -0.0068]
 [ 0.0097  0.0041]]
[1905. 1024.] 4
```

Check yourself

On paper first, then with your neighbour:

1. `ledger["V00"]["prices"][1:]`: what type, and what value?
2. Say in one sentence: `[t for t in ledger if ledger[t]["shares"] > 3]`.
3. What does `sorted(["b", "a", "C"], key=lambda s: s.lower())` return?
4. `prices[:, 0]`: which numbers? And `prices.sum(axis=1)`?
5. In pandas, why does `df[df["value"] > 1000 and df["shares"] > 3]` fail, and how do you fix it?

What you can now do

- Read nested data as a tree, and a chain of look-ups level by level.
- Read list, dict and set comprehensions, with and without a filter, and rewrite them as loops.
- Read `lambda`, `key=` and conditional expressions.
- Index and slice numpy arrays in two dimensions, with `axis`, broadcasting and masks.
- Build, select, filter, group and sort a pandas DataFrame, and tell `.loc` from `.iloc`.
- Say what every step of a method chain returns.